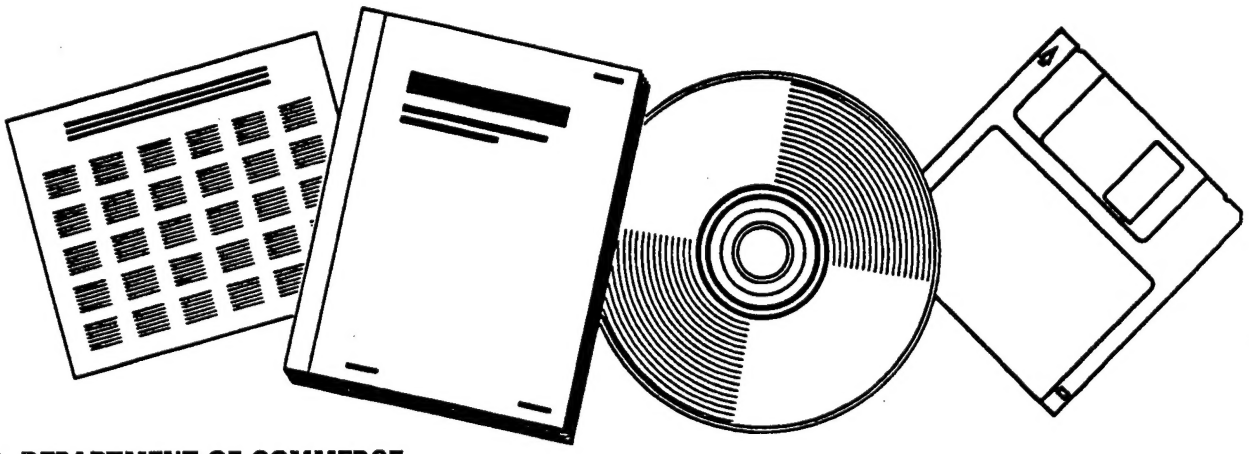
PB96-148549

PROCESSOR-EFFICIENT IMPLEMENTATION OF A MAXIMUM FLOW ALGORITHM

STANFORD UNIV., CA

JAN 90



U.S. DEPARTMENT OF COMMERCE
National Technical Information Service

January 1990

Report No. STAN-CS-90-1301

PB96-148549

Processor-Efficient Implementation of a Maximum Flow Algorithm

by

Andrew V. Goldberg

Department of Computer Science

Stanford University

Stanford, California 94305

Processor-Efficient Implementation of a Maximum Flow Algorithm

*Andrew V. Goldberg**
Department of Computer Science
Stanford University
Stanford, CA 94305

January 1990

*Research partially supported by NSF Presidential Young Investigator Grant CCR-8858097, IBM Faculty Development Award, a grant from 3M Corporation, and ONR Contract N00014-88-K-0166.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT: unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): STAN-CS-90-1301	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Department; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): Stanford University, Stanford, CA 94305	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING / SPONSORING ORGANIZATION: ONR; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N00014-88-K-0166
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Processor-Efficient Implementation of a Maximum Flow Algorithm

12. PERSONAL AUTHOR(S)
Andrew Goldberg

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
	FROM _____ TO _____	January 1990	10

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

In this paper we describe two processor-efficient implementation of the Maximum Distance Discharge algorithm for the maximum flow problem. Using $p = O(\sqrt{m})$ processors, the first implementation runs in $O(n^2 \log(2m/n + p)(\sqrt{m}/p))$ time and uses $O(m + n \log n)$ space; the second implementation runs in $O(n^2 \log n(\sqrt{m}/p))$ time and uses $O(m + p \log n)$ space. These bounds are within a logarithmic factor of the $O(n^2\sqrt{m})$ time and $O(m + n)$ space bounds on the sequential Maximum Distance Discharge Algorithm.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT: ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION
22a. NAME OF RESPONSIBLE INDIVIDUAL: Andrew Goldberg	22b. TELEPHONE (Include Area Code): (415) 723-2273; 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603

Abstract

In this paper we describe two processor-efficient implementation of the Maximum Distance Discharge algorithm for the maximum flow problem. Using $p = O(\sqrt{m})$ processors, the first implementation runs in $O(n^2 \log(2m/n + p)(\sqrt{m}/p))$ time and uses $O(m + n \log n)$ space; the second implementation runs in $O(n^2 \log n(\sqrt{m}/p))$ time and uses $O(m + p \log n)$ space. These bounds are within a logarithmic factor of the $O(n^2\sqrt{m})$ time and $O(m + n)$ space bounds on the sequential Maximum Distance Discharge Algorithm.

1 Introduction

The maximum flow problem is a classical combinatorial optimization problem, which has been widely studied in the context of sequential computation (see e.g. [7, 12]). Recently, parallel algorithms for the problem have been studied as well. Although the problem is known to be P-complete [14], significant speedups can be obtained by using a parallel algorithm for the problem, both in theory and in practice [11].

The first parallel algorithm for the maximum flow problem is due to Shiloach and Vishkin [18]. This algorithm is based on the blocking flow method of Dinic [6] and runs in $O(n^2 \log n)$ time using n processors and $O(n^2)$ memory.[1] In [10], the author introduced the *first-in, first-out* (*FIFO*) algorithm that runs in the same time and processor bounds but uses $O(m)$ of memory. This algorithm is the first of the *push-relabel* maximum flow algorithms. The *push-relabel* method was developed by Goldberg and Tarjan [13] as a generalization of it. The *maximum distance discharge* (*MDD*) algorithm [13] is another variation of the generic push-relabel method. A parallel implementation of this algorithm similar to that of the *FIFO* algorithm achieves the same asymptotic resource bounds [13].

The original sequential running time bound for both the *FIFO* and the *MDD* algorithms was $O(n^3)$. Cheriyan and Macheshwari [3] show that this bound is tight for the *FIFO* algorithm (*i.e.*, the algorithm requires $\Omega(n^3)$ time in the worst case), whereas for the *MDD* algorithm the bound can be improved to $O(n^2\sqrt{m})$. Thus an n-processor, $O(n^2 \log n)$-time parallel implementation is reasonable for *FIFO* algorithm since the corresponding time-processor product is within a logarithmic factor of the sequential time bound. However, such an implementation is not as good for the *MDD* algorithm, since the time-processor product exceeds the sequential running time bound by a factor of $O((n/\sqrt{m}) \log n)$, which is quite large for sparse graphs.

In this paper we describe an implementation of the *MDD* algorithm that runs in $O(n^2 \log n))$ time using $\sqrt{m}$ processors and $O(m + n \log n)$ memory. Using $p = O(\sqrt{m})$ processors, the implementation runs in $O(n^2 \log(2m/n + p)(\sqrt{m}/p))$ time. A variation of this implementation that uses the same number of processors and $O(m + p \log n)$ memory runs in $O(n^2 \log n(\sqrt{m}/p))$ time. These are the best strongly polynomial bounds for a processor-efficient maximum flow algorithm. (If the

[1] Throughout this paper, n and m denote the number of vertices and the number of arcs in the input network.

capacities are integers bounded by U, a parallel implementation of a scaling version of the *push-relabel* method, due to Ahuja and Orlin [1], runs in $O(n^2 \log(U) \log n)$ time using m/n processors and $O(m)$ memory.) The techniques and data structures used in our implementation may be useful for obtaining processor-efficient implementations of other graph algorithms.

Processor-efficient algorithms for the bipartite matching problem, which is closely related to the maximum flow problem, are discussed in [9].

2 Background

We use the following definitions. Let $G = (V, E)$ be a directed graph with vertex set V of size n and arc set E of size m. For ease in stating time bounds, we assume that $m > n$ and therefore $\log(m/n) > 0$. Define $E^{-1} = \{(w, v) | (v, w) \in E\}$ and $E^+ = E \bigcup E^{-1}$. For any vertex w we denote by $E(w)$ the set of vertices *adjacent out from* w, $E(w) = \{x | (w, x) \in E\}$, and by $E^{-1}(w)$ the set of vertices *adjacent into* w, $E^{-1}(w) = \{v | (v, w) \in E\}$. Graph G is a *network* if it has two distinguished vertices, a *source* s and a *sink* t, and a nonnegative real-valued capacity $u(v, w)$ on every arc (v, w). A *preflow* on a network is a nonnegative real-valued function f on the arcs such that $f(v, w) \leq u(v, w)$ for every arc (v, w) and $\sum_{v \in E^{-1}(w)} f(v, w) \geq \sum_{x \in E(w)} f(w, x)$ for every vertex $w \neq s$. The quantity $e_f(w) = \sum_{v \in E^{-1}(w)} f(v, w) - \sum_{x \in E(w)} f(w, x)$ is called the *excess* at vertex w. A preflow f is a *flow* if $e_f(w) = 0$ for every vertex $w \notin \{s, t\}$. A *cost function* $c : E \rightarrow \mathbf{R}$ assigns a cost to arcs of the network. We assume that costs are integers in the range $[-C, \ldots, C]$. The *residual capacity* of an arc (v, w) with respect to a preflow f is $u_f(v, w) = u(v, w) - f(v, w)$. Arc (v, w) is *saturated* if $u_f(v, w) = 0$ and *residual* if $u_f(v, w) > 0$. A *value* of a flow f is the excess of the sink $e_f(t)$. The *maximum flow* problem is to find a flow of the biggest value.

To get slightly better running time bounds, we sometimes assume, without loss of generality, that the maximum degree of a vertex in the input graph is at most $\Delta = 2m/n$. To justify this assumption, consider an arbitrary graph and replace each vertex v with degree $deg(v) > \Delta$ by $k = \lceil deg(v)/(\Delta - 2) \rceil$ vertices $v_1, \ldots, v_k$ connected in a ring with arcs of a very high capacity (e.g. the sum of all original capacities). Distribute arcs of v along $v_1, \ldots, v_k$ so that the degree of the new vertices is bounded by Δ. If the original graph has n vertices and m arcs, the transformed graph has at most $2n$ vertices and $m + n$ arcs.

Our model of parallel computation is the *concurrent-read, exclusive-write parallel random access machine* (CREW PRAM) [8]. We will use the fact that in this model, given a list of size l and $p \geq l$ processors, *ranking* the list, doing a *parallel prefix computation* on the list, and *sorting* the list takes $O(\log l)$ time [4, 5, 15, 17].

push(v, w).	
Applicability:	v is active **and** (v, w) is admissible.
Action:	send $\delta \in (0, \min(e_f(v), u_f(v, w))]$ units of flow from v to w.
relabel(v).	
Applicability:	either s or t is reachable from v in G_f **and** $\forall w \in V$ $u_f(v, w) = 0$ or $d(w) \geq d(v)$.
Action:	replace $d(v)$ by $\min_{(v,w)\in E_f}\{d(w)\} + 1$.

Figure 1: The *push* and *relabel* operations.

3 The Push and Relabel Operations

In this section we review the *push* and *relabel* operations. See [13] for more details.

To describe these operations, we need the following definitions. For a given preflow f, a *distance labeling* is a function d from the vertices to the nonnegative integers such that $d(t) = 0$, $d(s) = n$, and $d(v) \leq d(w) + 1$ for all residual arcs (v, w). We say that a vertex v is *active* if $v \notin \{s, t\}$ and $e_f(v) > 0$. Note that a preflow f is a flow if and only if there are no active vertices. An arc (v, w) is *admissible* if $(v, w) \in E_f$ and $d(v) = d(w) + 1$.

The *push-relabel* method maintains a preflow f and a distance labeling d, which are modified using the *push* and the *relabel* operations, respectively. A *push* from v to w increases $f(v, w)$ and $e_f(w)$ by $\delta = \min\{e_f(v), u_f(v, w)\}$, and decreases $f(w, v)$ and $e_f(v)$ by the same amount. The push is *saturating* if $u_f(v, w) = 0$ after the push and *nonsaturating* otherwise. A *relabel* operation, applied to a vertex v, sets the label of v equal to the largest value allowed by the valid labeling constraints. The *basic* operations are summarized in Figure 1.

The *generic* push-relabel method initializes f and d and repetitively performs an applicable *push* or *relabel* operation. When no operation applies, the method terminates. During initialization, f is set to the arc capacity on each arc leaving the source and zero on all arcs not incident to the source. The distance labeling is initialized as follows: $d(s) = n$ and $d(v) = 0$ for $v \in V - \{s\}$. A summary of the algorithm appears in Figure 2.

The generic method has the following properties [13]:

- The algorithm always terminates with a maximum flow.
- The number of *relabel* operations used is $O(n^2)$ and the total cost of these operations is $O(nm)$.
- The number of saturating *push* operations and the total cost of these operations is $O(nm)$.
- The number of nonsaturating *push* operations and the total cost of these operations is $O(n^2 m)$.

```
procedure generic (V, E, u);
    [initialization]
    ∀(v, w) ∈ E do begin
        f(v, w) ← 0;
        if v = s then f(s, w) ← u(s, w);
        if w = s then f(v, s) ← −u(s, v);
    end;
    ∀w ∈ V do begin
        e_f(w) ← Σ_{(v,w)∈E} f(v, w);
        if w = s then d(w) = n else d(w) = 0;
    end;
    [loop]
    while ∃ an active vertex do
        select an update operation and apply it;
    return(f);
end.
```

Figure 2: The generic maximum flow algorithm.

```
discharge(v).
Applicability: v is active.
Action:        while e_f(v) > 0 and v is not relabeled do
               if ∃ an admissible arc (v, w)
                 then push(v, w)
                 else relabel(v);
```

Figure 3: The discharge operation.

4 The Maximum Distance Discharge Algorithm

The generic algorithm does not specify the ordering in which the basic operations are applied. Some orderings, however, are more efficient then others. In this section we describe an ordering of the operations that leads to an $O(n^2\sqrt{m})$-time sequential algorithm. As we shall see later, this algorithm has a substantial degree of parallelism. Since parallel algorithms are of main concern here, we omit low-level detail of the sequential algorithm in our description. These details can be found in [12, 13].

The *discharge* operation, described in Figure 3, combines the basic operations locally (at a vertex). The *discharge* operation is applicable to an active vertex v. This operation iteratively reduces the excess at v by pushing it through admissible arcs going out of v if such arcs exist; otherwise, *discharge* relabels v. The operation stops when the excess at v is reduced to zero or v is relabeled. Note that *discharge* relabeles v only when the *relabel* operation applies.

The second step to an efficient ordering of basic operation is to restrict the order of processing of active vertices. The *MIDD* algorithm always selects for discharging an active vertex with the

```
procedure process-vertex;
    remove a vertex v from B_b;
    old-label ← d(v);
    discharge(v);
    add each vertex w made active by the discharge to B_{d(w)};
    if d(v) ≠ old-label then begin
        b ← d(v);
        add v to B_b;
    end
    else if B_b = ∅ then b ← b − 1;
end.
```

Figure 4: The *process-vertex* procedure.

largest label. The corresponding parallel algorithm processes all such vertices at once.

The sequential implementation of the largest-label algorithm maintains an array of sets B_i, $0 \leq i \leq 2n-1$, and an index b into the array. Set B_i consists of all active vertices with label i, represented as a doubly-linked list, so that insertion and deletion take $O(1)$ time. The index b is the largest label of an active vertex. During the initialization, active vertices are placed in B_0, and b is set to 0. At each iteration, the algorithm removes a vertex from B_b, processes it using the *discharge* operation, and updates b. The algorithm terminates when b becomes negative, *i.e.*, when there are no active vertices. This processing of vertices, which implements the *while* loop of the generic algorithm, is described in Figure 4.

We define *a phase* of the algorithm as a maximal time interval during which b remains constant. The notion of phase is important both for the sequential and the parallel analysis of the algorithm.

Lemma 4.1 [13] The number of phases during an execution of the *MDD* algorithm is $O(n^2)$.

The following theorem gives the sequential running time bound for the algorithm.

Theorem 4.2 [3] The *MDD* algorithm runs in $O(n^2\sqrt{m})$ time.

Lemma 4.1 suggests a parallel version of the algorithm, where all largest-labeled active vertices are processes in parallel. The running time of the resulting algorithm is $O(n^2)$ times the time needed for the parallel processing of the vertices. The next section describes such an implementation.

5 A Processor-Efficient Parallel Implementation

Straight-forward implementations of parallel maximum flow algorithms, described in [11, 18], use a linear number of processors. Shiloach and Vishkin [18] show that their algorithm can be implemented with n processors and $O(n^2)$ space and still achieve the $O(n^2 \log n)$ time bound. In this

section we extend their techniques to obtain a parallel implementation of the *MDD* algorithm that uses $\sqrt{m}$ processors, runs in $O(n^2 \log n)$ time, and needs $O(m + n \log n)$ space. Using $p = O(\sqrt{m})$ processors, the implementation runs in $O(n^2 \log(\Delta+p))(\sqrt{m}/p)$ time. The time-processors product of this parallel implementation is within a logarithmic factor of the number of operations of the underlying sequential method. A variation of this implementation achieves a slightly better space bound at the expense of a slightly worse time bound.

To obtain a processor-efficient implementation of the algorithm, we have to provide a mechanism for assigning the work to processors in such a way that most processors are busy most of the time. Doing this scheduling "on-line" is the biggest problem our implementation has to overcome.

The implementation maintains the sets B_i of active vertices with the distance label i, for $0 \leq i \leq 2n-1$. The index b is maintained as in the sequential implementation. The sets are maintained so that in $O(\log p)$ time, several processors can add a vertex each to the sets, and several elements of B_b can be assigned to different processors and removed from the set.

A straight-forward way to implement these operations is to use an array of length n for each set. The elements of B_i occupy the first $|B_i|$ locations of the corresponding array. The processors that want to add elements to $|B_i|$ are enumerated and add their elements to the array position determined by their rank and $|B_i|$; after this is done, $|B_i|$ is updated. To assign elements of B_b to a set of processors of size $k \leq B_b$, the processors are ranked and assigned elements starting from the end of the corresponding array. Then B_b is decreased by k. This straightforward implementation meets the desired time bound but uses $O(n^2)$ space.

To reduce the space requirement, we take advantage of the fact that the total number of elements in all sets B_i is at most n. We discuss two parallel data structures that can be used to maintain sets B_i in a space-efficient way.

One such data structure is a *dynamic array*. A dynamic array consists of an ordered list of segments. If the dynamic array contains k elements, the number of segments in the list is $\lceil \log k \rceil + 1$. Each segment is an array; the length of the first two segments is 1, and for $j > 2$, the length of the jth segment is 2^{j-1}. Note that more than half of the space allocated to the nonempty segments is actually used and all segment are of size at most n. Using this observation, it is easy to see that the total space required is $O(n \log n)$.

We can implement the lists of segments by arrays of pointers to the segments. These arrays take $O(n \log n)$ space. The time required for l processors to add elements to the sets B_i or to remove l elements from B_b is $O(\log l)$ (using sorting or ranking of processors operating on the same set).

Alternatively, we can use the parallel 2-3 tree data structure described in [16]. This data structure allows addition (deletion) of l elements by l processors to (from) a set of k elements in $O(\log k + \log l)$ time and $O(k + l \log k)$ space. In our application $k \leq n$ and $l \leq p = O(n)$, so the bounds can be rewritten as $O(\log n)$ time and $O(n + p \log n)$ space.

Our implementation of the MDD algorithm works in iterations, each of which implements a

pass of the sequential algorithm. Each iteration is divided into three phases. During the first phase flow is pushed out of the active vertices in B_b. During the second phase this flow is collected at the destination vertices. The last phase relabels the appropriate vertices.

For the purpose of scheduling, one has to keep track of the number of processors needed to perform a *relabel* or a *discharge* operation. In the sequential algorithm, the number steps required to relabel a vertex v is linear in the degree of v, so in the parallel implementation we assign for this operation the number of processors equal to the degree of v. Discharging a vertex v requires the number of processors equal to the number of pushes performed during the discharge. We maintain a data structure at each vertex that allows fast computation of this number by a single processor. This data structure is also used for pushing the flow.

The data structure we use is a variant of the *partial sum tree* data structure [18]. A partial sum tree is a balanced binary tree with leaves corresponding to edges adjacent to a vertex. Each vertex v has two trees associated with it, the *out-tree*(v) which is used to push flow out of the vertex, and the *in-tree*(v) which is used to collect flow pushed into the vertex.

The *out-trees* are used in the first phase. Leaves of the *out-tree*(v) correspond to the arcs (v, w). Each node x of the tree has two labels, $a(x)$ and $b(x)$. The label values are defined as follows. If x is a leaf corresponding to the arc (v, w), then

$$a(x) = \begin{cases} u_f(v, w)) & \text{if } (v, w) \text{ is admissible} \\ 0 & \text{otherwise} \end{cases}$$

and

$$b(x) = \begin{cases} 1 & \text{if } (v, w) \text{ is admissible} \\ 0 & \text{otherwise.} \end{cases}$$

If x is not a leaf, then $a(x)$ and $b(x)$ are equal to the sums of the corresponding values of the children of x. In other words, $a(x)$ is equal to the sum of residual capacities of the admissible arcs corresponding to the leaves of the subtree rooted at x, and $b(x)$ is the number of such admissible arcs.

Suppose v is an active vertex to be discharged. First a processor is assigned to v to determine the number of pushes $p(v)$ which will be made out of v. Using a and b values of *out-tree*(v) and the value of $e_f(v)$, this can be done in $O(\log \Delta)$ time.

To push the flow out of v, we assign $p(v)$ processors to v. Then we associate each processor with the arc it will push the flow through. To do this, we rank the processors assigned to v, which takes $O(\log p)$ time. Then each processor goes down the tree starting from the root and picking at each step the left or the right child of its current node x depending on the processor's rank and on the b values of the children of x. At the end, the ith processor will be at the leaf corresponding to the ith admissible arc of v. Note that this process requires concurrent read. Then each of the processors computes the amount $\delta(v, w)$ to be pushed along the arc corresponding to the processor. For all but the last processor assigned to v, this amount is equal to $u_f(v, w)$, since

the corresponding pushes are saturating. For the last processor, this amount is equal to $u_f(v,w)$ if $a(\textit{root}(\textit{out-tree}(v))) \leq e_f(v)$ and to $a(\textit{root}(\textit{out-tree}(v))) - e_f(v)$ otherwise. The processors update the flow function on the corresponding arcs and the last processor updates $e_f(v)$.

Next the *out-trees* are updated going from the bottom level of the tree up. Initially each processor updates the a and b labels at the leaf assigned to it. Then the processor decides if it will stop updating or not. The processor stops the update only if it is currently at the root of the tree or if it is at a tree node x which is a right son of its father y, and the left son of y has just been updated, *i.e.*, also has a processor working on it. The process is repeated until the root of the tree is reached and updated.

In the second stage, the flow pushed into vertices w is collected using the in-trees. Leaves of $\textit{in-tree}(w)$ correspond to arcs entering w. A processor that was assigned to the arc (v,w) when pushing flow out of v is assigned to the same arc when processing the flow pushed into w. Every node x of $\textit{in-tree}(w)$ has a variable $a'(x)$ associated with it. If x is a leaf corresponding to an arc (v,w), then $a'(x)$ is set to the amount equal to that just pushed along the arc. If x is not a leaf, then $a'(x)$ is equal to the sum of the values of the corresponding variables of its children. The values of a' variables are propagated going from the leaves to the root in the same way as the values of a variables of the out-trees. The update takes $O(\log \Delta)$ time. After the update, $e_f(w)$ is increased by $a'(\textit{in-tree}(\textit{root}(v)))$. Then the values of a' variables are reinitialized to 0 by making another leaves-to-root pass.

Relabelings are implemented by maintaining an array of vertices to be relabeled. (Note that this array has at most n items). Vertices that were unable to get rid of their excesses during a discharge are added to the end of the array using ranking of the processors that want to add the vertices to the array. During the relabeling stage, processors are assigned to the last p elements of the array or to every element of the array if there are less then p elements in it. Using parallel prefix computations on the portion of the array for which the processors have been assigned, vertices needing relabeling are assigned the number of processors equal to their degrees, and the relabeling is performed by doing a parallel prefix computation on edge list of every vertex that needs to be relabeled.

The analysis is based on the following theorem of Brent.

Theorem 5.1 [2] Any synchronized parallel algorithm of depth d that consists of a total of x elementary operations can be implemented by p processors within a depth of $\lceil x/p \rceil + d$.

Let *macro-operations* be any standard unit-time PRAM operation plus sequences of operations performed by individual processors while working on an *in-tree* or an *out-tree* or performing an operation of ranking, sorting, or computing parallel prefix. Note that the macro-operations used by the algorithm take $O(\log(\Delta+p))$ time. By Lemma 4.1, the depth of the algorithm is $O(n^2)$ macro-operations. The total number of macro-operations used by the algorithm is $O(n^2\sqrt{m})$. Applying Theorem 5.1 for $p = O(\sqrt{m})$ and using the fact that each macro-operation takes $O(\log(\Delta + p))$

time, we get the following result.

Theorem 5.2 The parallel implementations of the *MDD* algorithm run in $O(n^2 \log(2m/n+p)(\sqrt{m}/p))$ time using $p = O(\sqrt{m})$ processors and $O(m + n \log n)$ memory, or in $O(n^2 \log n(\sqrt{m}/p))$ time using $p = O(\sqrt{m})$ processors and $O(m + p \log n)$ memory.

Note that the time bounds in the above theorem exceed those with optimal processor utilization by a factor of $O(\log n)$. The amount of space required by the implementations is (slightly) superlinear. The latter fact is due to the space requirements of the dynamic array or parallel 2-3 tree data structures.

We conclude with the following open question: Can the MDD algorithm be implemented to run in $O(n^2 \log n(\sqrt{m}/p))$ time using $p = O(\sqrt{m})$ processors and a *linear* amount of memory?

Acknowledgments

I would like to thank Robert Kennedy, Serge Plotkin, and Bob Tarjan for useful comments on the results presented in this paper. I also would like to thank Uzi Vishkin for bringing the parallel 2-3 tree data structure to my attention.

References

[1] R. K. Ahuja and J. B. Orlin. A Fast and Simple Algorithm for the Maximum Flow Problem. Sloan Working Paper 1905-87, Sloan School of Management, M.I.T., 1987.

[2] R. P. Brent. The Parallel Evaluation of General Arithmetic Expressions. *J. Assoc. Comput. Mach.*, 21:201–206, 1974.

[3] J. Cheriyan and S. N. Maheshwari. Analysis of Preflow Push Algorithms for Maximum Netwrok Flow. *SIAM J. Comput.*, 18:1057–1086, 1989.

[4] R. Cole. Parallel merge sort. In *Proc. 27th IEEE Annual Symposium on Foundations of Computer Science*, pages 511–516, 1986.

[5] R. Cole and U. Vishkin. Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. *Information and Control*, 70:32–53, 1986.

[6] E. A. Dinic. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[7] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NJ., 1962.

[8] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proc. 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[9] H. N. Gabow and R. E. Tarjan. Almost-Optimal Speed-ups of Algorithms for Matching and Related Problems. In *Proc. 20th Annual ACM Symposium on Theory of Computing*, pages 514–527, 1988.

[10] A. V. Goldberg. A New Max-Flow Algorithm. Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., 1985.

[11] A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers.* PhD thesis, M.I.T., January 1987. (Also available as Technical Report TR-374, Lab. for Computer Science, M.I.T., 1987).

[12] A. V. Goldberg, É. Tardos, and R. E. Tarjan. Network Flow Algorithms. Technical Report STAN-CS-89-1252, Department of Computer Science, Stanford University, 1989.

[13] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988.

[14] L. M. Goldschlager, R. A. Shaw, and J. Staples. The Maximum Flow Problem is Log Space Complete for P. *Theoretical Computer Sci.*, 21:105–111, 1982.

[15] R. E. Ladner and M. J. Fischer. Parallel Prefix Computation. *J. Assoc. Comput. Mach.*, 27:831–838, 1980.

[16] W. Paul, U. Vishkin, and H. Wagener. Parallel Dictionary. In *Proc. 10th International Colloquium on Automata, Languages and Programming*, pages 597–609. Springer-Verlag, 1983.

[17] J. T. Schwartz. Ultracomputers. *ACM Trans. Prog. Lang. and Syst.*, 2:484–521, 1980.

[18] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ Parallel Max-Flow Algorithm. *J. Algorithms*, 3:128–146, 1982.

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Technical Information Service
Springfield, VA 22161 (703) 487-4650